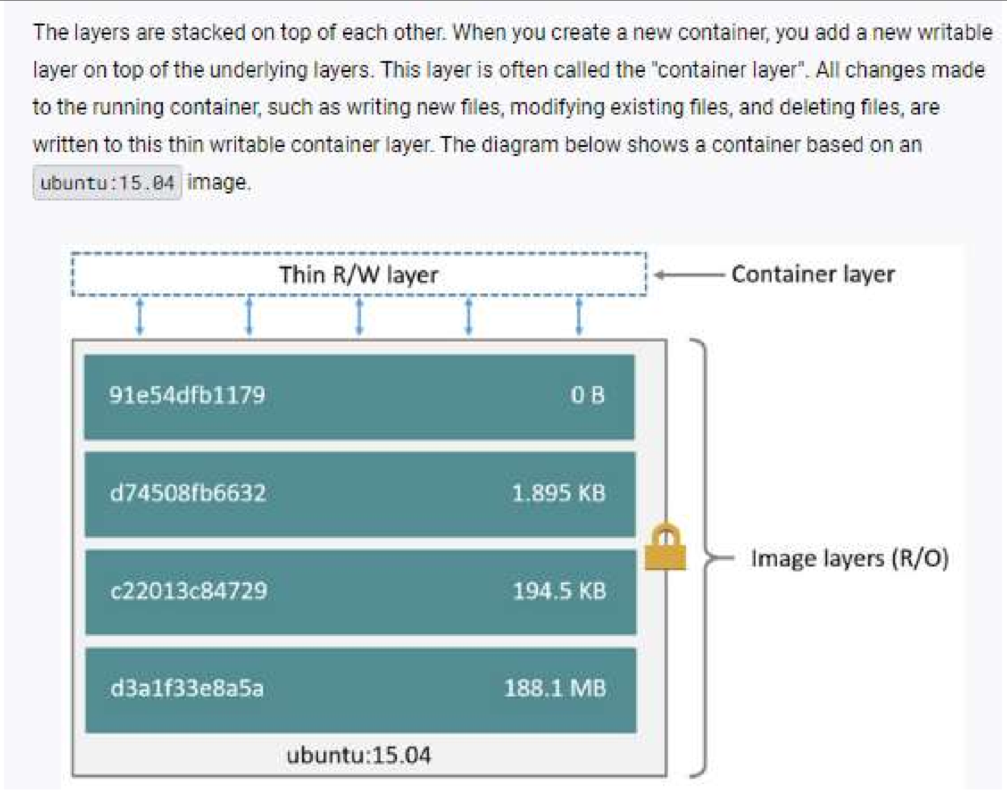
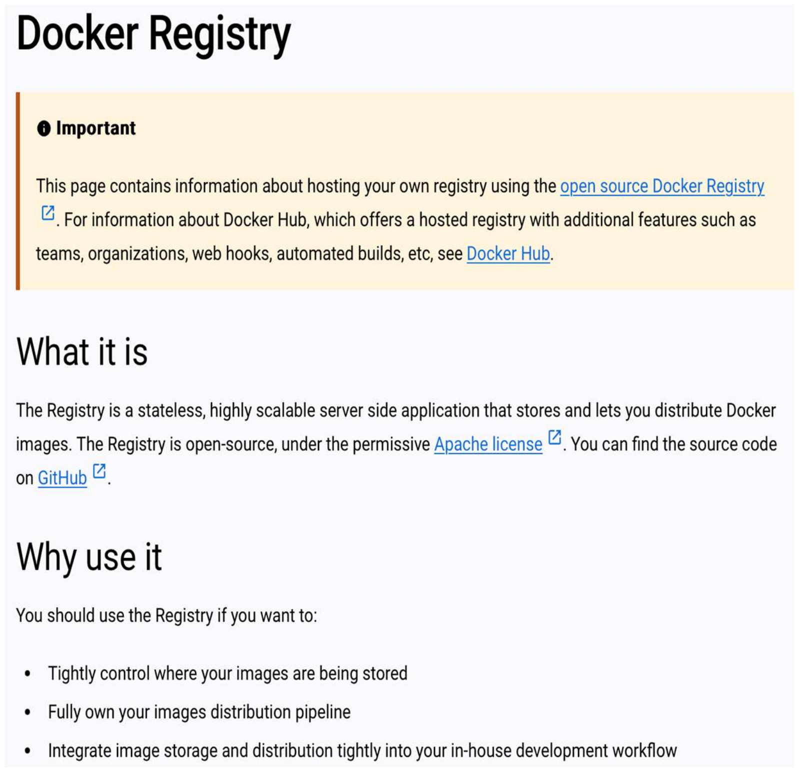
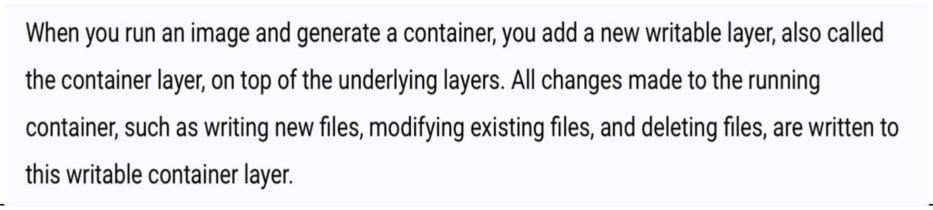
Exhibit 6 to Complaint
Intellectual Ventures I LLC and Intellectual Ventures II LLC

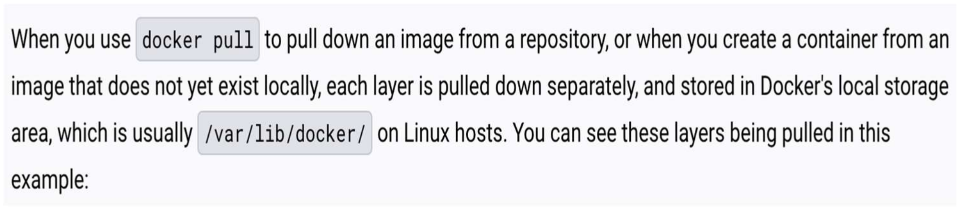
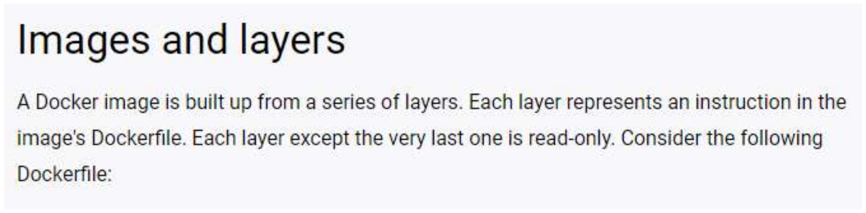
Example Comerica Count I Systems and Services
U.S. Patent No. 8,332,844 (“the ’844 Patent”)

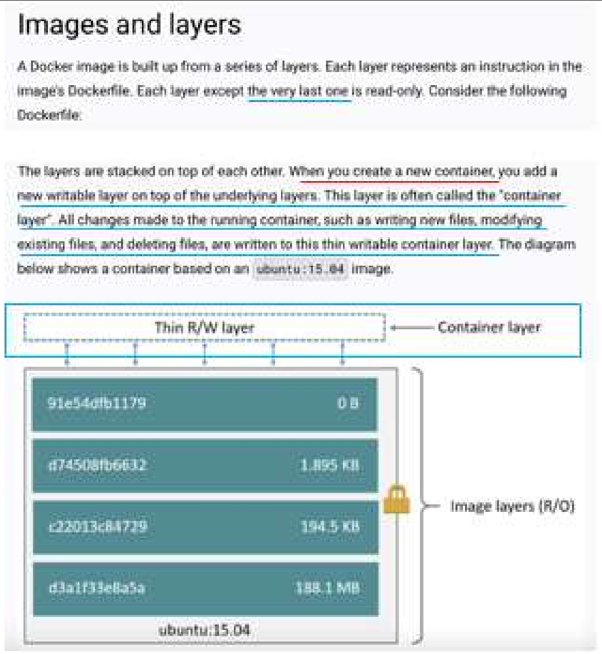
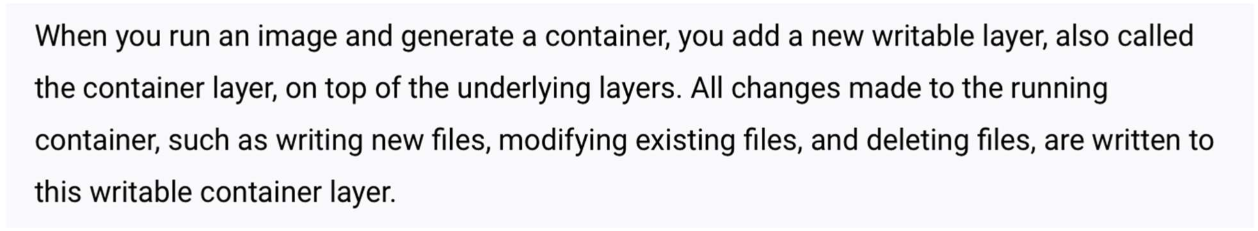
The Accused Systems and Services include, without limitation, Comerica’s systems that utilize Docker; all past, current, and future systems and services that operate in the same or substantially similar manner as the specifically identified systems and services; and all past, current and future Comerica’s systems and services that have the same or substantially similar features as the specifically identified systems and services (“Example Comerica Count I Systems and Services”).

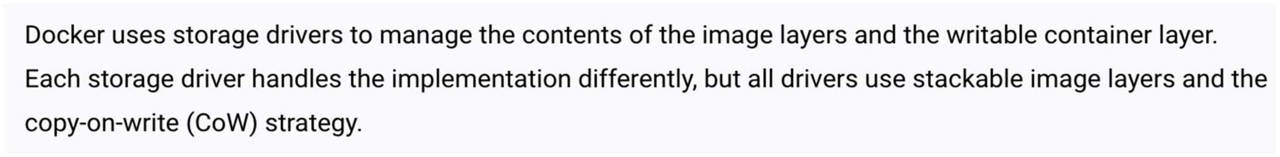
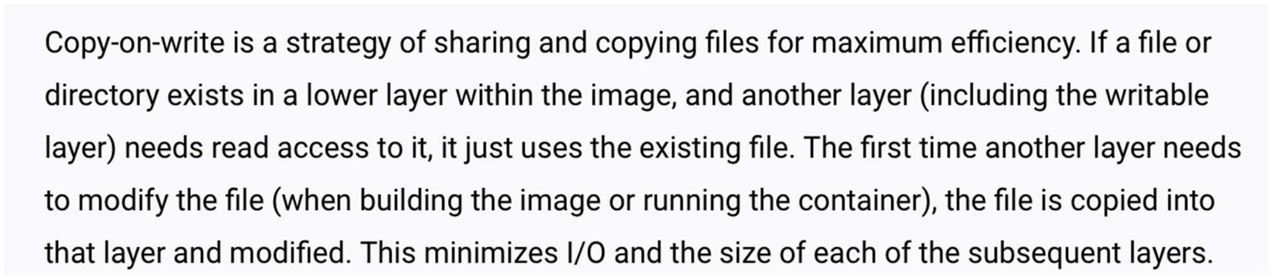
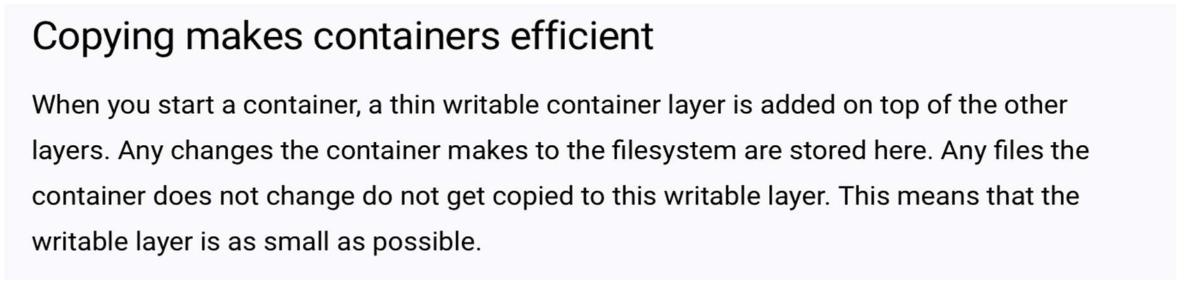
U.S. Patent No. 8,332,844	
Example Claim 7	**Example Comerica Count I Systems and Services**
7. A method for providing data to a plurality of compute nodes, comprising:	Upon information and belief, Comerica's systems perform "a method for providing data to a plurality of compute nodes." Docker provides docker images to a compute node in a cluster. *See* https://www.linkedin.com/in/michaelmxu/ (last accessed on November 13, 2023) (noting the use of Docker at Comercia).
storing blocks of a root image of said compute nodes on a first storage unit;	Upon information and belief, Comerica's systems perform "storing blocks of a root image of said compute nodes on a first storage unit." Docker stores blocks of a root image (via the collection of read-only layers in a Docker image) in the Docker registry.

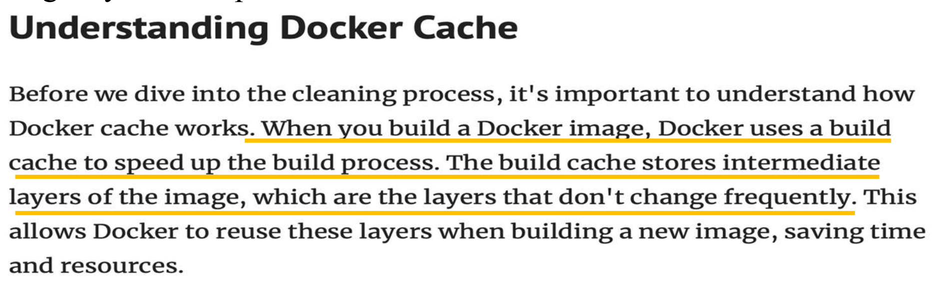
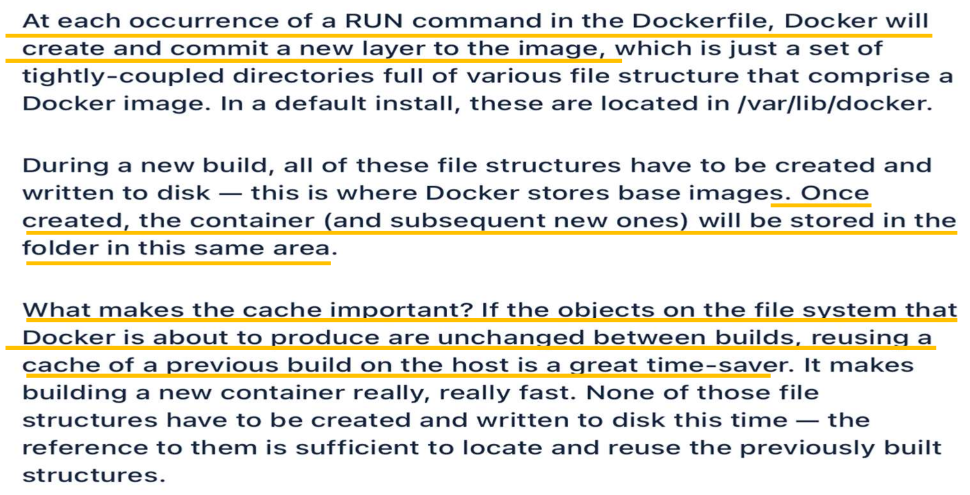
	 The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image. Thin R/W layer Container layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 Image layers (R/O) *See* https://docs.docker.com/storage/storagedriver/ (last accessed on November 9, 2023).

	 Docker Registry Important This page contains information about hosting your own registry using the open source Docker Registry. For information about Docker Hub, which offers a hosted registry with additional features such as teams, organizations, web hooks, automated builds, etc, see Docker Hub. What it is The Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images. The Registry is open-source, under the permissive Apache license. You can find the source code on GitHub. Why use it You should use the Registry if you want to: • Tightly control where your images are being stored • Fully own your images distribution pipeline • Integrate image storage and distribution tightly into your in-house development workflow *See* https://docs.docker.com/registry/ (last accessed on November 9, 2023).
storing leaf images for respective compute nodes on respective second storage units,	Upon information and belief, Comerica's systems perform "storing leaf images for respective compute nodes on respective second storage units." Docker stores leaf images (thin R/W layer in a runnable container) for respective compute nodes on respective second storage units (Docker's local storage area).  When you run an image and generate a container, you add a new writable layer, also called the container layer, on top of the underlying layers. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this writable container layer.

	See https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (last accessed on November 9, 2023).  When you use `docker pull` to pull down an image from a repository, or when you create a container from an image that does not yet exist locally, each layer is pulled down separately, and stored in Docker's local storage area, which is usually `/var/lib/docker/` on Linux hosts. You can see these layers being pulled in this example: *See* https://docs.docker.com/storage/storagedriver/ (last accessed on November 9, 2023).  Images and layers A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

	 Images and layers A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile: The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image. Thin R/W layer Container layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 Image layers (R/O) *See* https://docs.docker.com/storage/storagedriver/ (last accessed on November 9, 2023).
said leaf images including only additional data blocks not previously contained in said root image and changes made by respective compute nodes to the blocks of the root image,	Upon information and belief, Comerica's systems also have "said leaf images including only additional data blocks not previously contained in said root image and changes made by respective compute nodes to the blocks of the root image." Docker includes a writable per runnable container layer for new data such as new files and including a copy on write strategy for changes made to the blocks of the root image.  When you run an image and generate a container, you add a new writable layer, also called the container layer, on top of the underlying layers. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this writable container layer.

	See https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ (last accessed on November 9, 2023).  Docker uses storage drivers to manage the contents of the image layers and the writable container layer. Each storage driver handles the implementation differently, but all drivers use stackable image layers and the copy-on-write (CoW) strategy.  Copy-on-write is a strategy of sharing and copying files for maximum efficiency. If a file or directory exists in a lower layer within the image, and another layer (including the writable layer) needs read access to it, it just uses the existing file. The first time another layer needs to modify the file (when building the image or running the container), the file is copied into that layer and modified. This minimizes I/O and the size of each of the subsequent layers. *See* https://docs.docker.com/storage/storagedriver/ (last accessed on November 9, 2023).
wherein said leaf images of respective compute nodes do not include blocks of said root image that are unchanged by respective compute nodes; and	Upon information and belief, Comerica's systems include "wherein said leaf images of respective compute nodes do not include blocks of said root image that are unchanged by respective compute nodes." Docker contains a writable layer using copy on write, leaving unchanged image data in their original image layers.  Copying makes containers efficient When you start a container, a thin writable container layer is added on top of the other layers. Any changes the container makes to the filesystem are stored here. Any files the container does not change do not get copied to this writable layer. This means that the writable layer is as small as possible.

	See https://docs.docker.com/storage/storagedriver/ (last accessed on November 9, 2023).
caching blocks of said root image that have been accessed by at least one of said compute nodes in a cache memory.	Upon information and belief, Comerica's systems perform "caching blocks of said root image that have been accessed by at least one of said compute nodes in a cache memory." Docker caches unchanged image layers from previous RUN commands.  **Understanding Docker Cache** Before we dive into the cleaning process, it's important to understand how Docker cache works. When you build a Docker image, Docker uses a build cache to speed up the build process. The build cache stores intermediate layers of the image, which are the layers that don't change frequently. This allows Docker to reuse these layers when building a new image, saving time and resources. *See* https://collabnix.com/how-to-clear-docker-cache/ (last accessed on November 9, 2023).  At each occurrence of a RUN command in the Dockerfile, Docker will create and commit a new layer to the image, which is just a set of tightly-coupled directories full of various file structure that comprise a Docker image. In a default install, these are located in /var/lib/docker. During a new build, all of these file structures have to be created and written to disk — this is where Docker stores base images. Once created, the container (and subsequent new ones) will be stored in the folder in this same area. What makes the cache important? If the objects on the file system that Docker is about to produce are unchanged between builds, reusing a cache of a previous build on the host is a great time-saver. It makes building a new container really, really fast. None of those file structures have to be created and written to disk this time — the reference to them is sufficient to locate and reuse the previously built structures. *See* https://thenewstack.io/understanding-the-docker-cache-for-faster-builds/ (last accessed on November 9, 2023).